# Perl Wrappers for HDF5 APIs

# User's Guide

**The HDF Group**
January, 2008

## *Introduction*

HDF5 is a widely used portable file format and library for storing, retrieving, analyzing, visualizing and converting data. HDF5 stores multidimensional arrays along with metadata in a portable file format. It supports hierarchical and other organizing structures, providing users with a high degree of flexibility for organizing and managing data.

The HDF5 library provides several interfaces, or APIs. These APIs provide routines for creating, accessing, and manipulating HDF5 files and objects; the library itself is implemented in C. To facilitate the work of FORTRAN 90, C++, and Java programmers, HDF5 function wrappers have been developed in each of these languages. An implementation of wrappers for Perl has been developed as a prototype that will not be part of the standard HDF5 distribution. Those interested in the Perl prototype are encouraged to contribute corrections, additions, and comments.

Efforts to provide HDF5 support for Perl are motivated by the growing interested in HDF5 within the bioinformatics community, where Perl is a very common language. The features illustrated in this work are the ones that we have found to be of potential value dealing with data related to DNA sequencing.

This document describes the installation of the Perl wrappers, HDF5 files and datasets, and read/write operations. Additional topics include dataset extendibility and compression by using chunking storage, and partial write operations. The material should prepare the reader to analyze the Perl script presented in the Appendix, which migrates nucleic acid or peptide sequence data from FASTA format into HDF5 files.

## *Installation*

A prerequisite for installing the Perl wrappers is the availability of binaries for the HDF5 1.6.5 library.

The Perl wrappers can be installed using the following steps:

1. Modify Makefile.PL so that it points to the location of the HDF5 library.
2. perl Makefile.PL LIB=/install_path
3. make
4. make install

To use the wrappers, each program should contain the following lines at the beginning:

```
#!/usr/bin/perl

# load appropriate modules
use lib "/install_path";
use HDFPerl;
use strict;
use Init;

# initialize HDF5 constants
Init::initialize();
```

These lines load the appropriate modules and initialize HDF5 constants necessary to create and access HDF5 files and datasets.

## *HDF5 Files and Datasets*

The following is a brief introduction to HDF5. For more details, see the HDF5 documentation at http://www.hdfgroup.org/HDF5/doc/.

As suggested by the name Hierarchical Data Format, an HDF5 file can be hierarchically structured. The HDF5 groups and datasets implement this hierarchy.

In the simple and most common case, the file structure is a tree structure as shown in Figure 1; in the general case, the file structure may be a directed graph with a designated entry point. The tree structure is very similar to the file system structures employed on UNIX systems, directories and files, and on Apple Macintosh and Microsoft Windows systems, folders and files. HDF5 groups are analogous to the directories and folders; HDF5 datasets are analogous to the files. Thus, groups provide a way to organize objects within an HDF5 file, while datasets contain the application data.

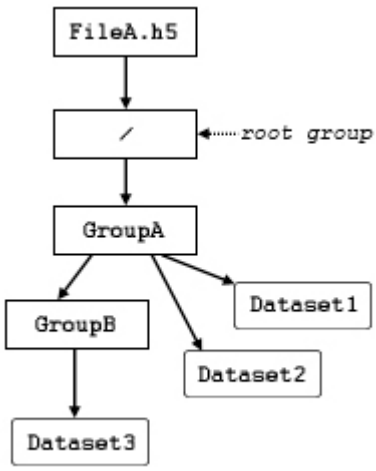


**Figure 1 An HDF5 file with a strictly hierarchical group structure**

An HDF5 dataset is an object composed of a collection of data elements, or raw data, and metadata that stores a description of the data elements, data layout, and all other information necessary to write, read, and interpret the stored data. From the viewpoint of the application the raw data is stored as a one-dimensional or multi-dimensional array of elements (the *raw data*), those elements can be of any of several numerical or character types, small arrays, or even compound types similar to database records.

A dataset may also include *attributes,* which are small metadata objects defined by applications to describe the nature and/or intended use of the dataset.

The following steps are required to create an HDF5 file, a group, a dataset, and to perform access to the dataset:

```
File creation
Specify the file creation and access property lists, if necessary.
Create the file.

      Group creation
      Obtain the location identifier where the group is to be
      created.
      Create the group.

      Dataset creation
      Obtain the location identifier where the dataset is to be
      created.
      Define a datatype (integer, float, char, …).
      Define a dataspace (array characteristics such as rank and
      size).
      Specify the dataset creation property list, if necessary
      Create the dataset.

                  Read / Write operation on dataset
                  Obtain the dataset identifier.
                  Specify the memory datatype.
                  Specify the memory dataspace.
                  Specify the file dataspace.
                  Specify the transfer property list, if necessary.
                  Perform the desired operation on the dataset.

      Close the dataspace, datatype, and property list if necessary.
      Close the dataset.

      Close the group.

Close the file and close the property lists, if necessary.
```

As an introductory example, we will create an HDF5 file, two groups, and a dataset inside one group. The dataset consists of a one-dimensional array of 10 integers. The file structure and the Perl script are shown in Figure 2 and List 1, respectively.

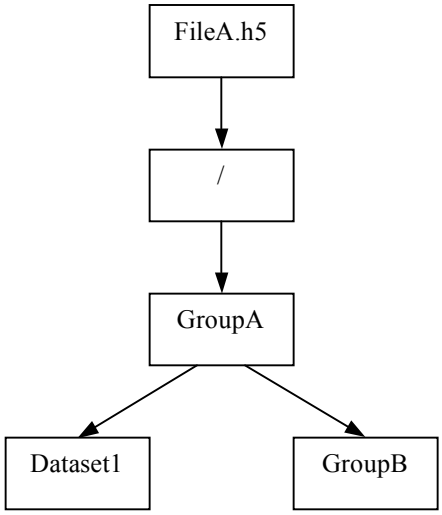


**Figure 2 HDF5 structure of sample script**

```
#!/usr/bin/perl

# load appropriate modules
use lib "/install_path";
use HDFPerl;
use strict;
use Init;

# initialize HDF5 constants
Init::initialize();

# create a new HDF File.
my $file_id = HDFPerl::h5fcreate_p("FileA.h5", $Init::H5F_ACC_TRUNC,
      $Init::H5P_DEFAULT, $Init::H5P_DEFAULT);

# create a group named "GroupA" under the root group
my $groupa_id = HDFPerl::h5gcreate_p($file_id, "GroupA", 0);

# create a group named "GroupB" under "GroupA"
my $groupb_id = HDFPerl::h5gcreate_p($groupa_id, "GroupB", 0);

# creates a dataspace: a one-dimensional array of 10 elements
my @dims=(10);
my $dspace_id = HDFPerl::h5screate_p(1, \@dims);

# creates a dataset in "GroupA" with the specified dataspace. Each
# element is an integer
my $dset_id = HDFPerl::h5dcreate_p($groupa_id, "Dataset1",
      $Init::H5T_NATIVE_INT, $dspace_id, $Init::H5P_DEFAULT);

# memory buffer containing data to be written in dataset
my @write_buf = (1,2,3,4,5,6,7,8,9,10);

# write entire dataset with buffer data
HDFPerl::h5dwrite_int_p($dset_id, $Init::H5T_NATIVE_INT,
```

```
$dspace_id, $dspace_id, $Init::H5P_DEFAULT, \@write_buf);
```

```
# read back entire dataset
my $read_ref = HDFPerl::h5dread_int_p($dset_id, $dspace_id, $dspace_id,
      $Init::H5P_DEFAULT);

# dereference reading array and print data
my @read_buf=@{$read_ref};
print "Read data: @read_buf\n";

# close dataset
HDFPerl::h5dclose_p($dset_id);

# close dataspace
HDFPerl::h5sclose_p($dspace_id);

# close groups
HDFPerl::h5gclose_p($groupa_id);
HDFPerl::h5gclose_p($groupb_id);

# close file
HDFPerl::h5fclose_p($file_id);
```

**List 1 Sample script showing basic HDF5 operations**

The script starts with a few standard initial lines that set the environment, load the appropriate modules, and initialize HDF5 constants. Then, the HDF5 file is created using the function `HDFPerl::h5fcreate_p`. This function specifies a filename and a file access mode (`$Init::H5F_ACC_TRUNC` will be appropriate for most cases). Arguments known as property lists that control how the file is created and accessed are set to the default `$Init::H5P_DEFAULT`.

Two groups are created using the function `HDFPerl::h5gcreate_p`. The function states the location of the groups (a file identifier corresponds to the root group) and their names. An additional hint parameter can be set to zero.

Before creating a dataset, its array structure must be defined in the form of a dataspace by using the function `HDFPerl::h5screate_p`. The arguments set the number of dimensions and the size of each dimension.

A dataset can then be created using the function `HDFPerl::h5dcreate_p` specifying the location of the dataset, its name, the datatype of each element, the dataspace, and a creation property list. Since we want to create an array of integer elements, we use the predefined HDF5 datatype `$Init::H5T_NATIVE_INT`. Examples of additional datatypes are shown in Figure 3. Although we can define a dataset creation property list, we used the default `$Init::H5P_DEFAULT` in this example.

| HDF5 Native Datatype | Datatype |
|---|---|
| `$Init::H5T_NATIVE_INT` | Integer |
| `$Init::H5T_NATIVE_FLOAT` | Floating point |
| `$Init::H5T_NATIVE_CHAR` | Character |
| `$Init::H5T_C_S1` | Character string |

**Figure 3 Examples of HDF5 predefined native datatypes**

After defining the memory buffer containing the integer data, the write operation is performed by using the function `HDFPerl::h5dwrite_int_p` which specifies the dataset identifier and datatype. The next two arguments correspond to identifiers of the memory dataspace and file dataspace, respectively. Because both dataspaces are identical in this example, we can use the same identifier for both arguments. A default transfer property list and a reference for the memory buffer are also specified.

The integer data can be read back from the dataset by using the function `HDFPerl::h5dread_int_p`. The arguments indicate the dataset, memory dataspace, file dataspace, and a transfer property list. As this function returns a reference for the reading buffer, dereferencing should be performed in order to use the buffer as a conventional data array.

At the end of the script, the used resources (dataset, dataspace, groups, and file) are closed using the appropriate functions.

Although the script prints the dataset elements on the screen, the contents and structure of the HDF5 file can also be visualized using the tools *HDFView* or *h5dump*[1]. The following is the output of *h5dump* applied on the file. Note that it corresponds to the structure shown in Figure 2.

[1] *HDFView* is a graphical viewer for HDF5 files. *h5dump* displays the contents of an HDF5 file in a text format. *HDFView, h5dump* and other tools are available at the THG web site, and are described at http://www.hdfgroup.org/products/hdf5_tools/.

```
HDF5 "FileA.h5" {
GROUP "/" {
   GROUP "GroupA" {
      DATASET "Dataset1" {
         DATATYPE  H5T_STD_I32LE
         DATASPACE  SIMPLE { ( 10 ) / ( 10 ) }
         DATA {
         (0): 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
         }
      }
   }
   GROUP "GroupB" {
   }
}
}
```

## Remarks

- `HDFPerl::h5fcreate_p` creates an HDF5 file and returns the file identifier.

```
file_id HDFPerl::h5fcreate_p(name, access_mode,
      creation_plist, access_plist)
```

  - The *name* parameter specifies the name of the file to be created.
  - The *access_mode* parameter specifies the file access mode. `$Init::H5F_ACC_TRUNC` will truncate a file if it already exists.
  - The *creation_plist* parameter specifies the file creation property list. Using `$Init::H5P_DEFAULT` indicates that the default file creation property list is to be used.
  - The *access_plist* parameter specifies the file access property list. Using `$Init::H5P_DEFAULT` indicates that the default file creation property list is to be used.
  - This function returns the file identifier if successful and a negative value otherwise.

- When a file is no longer accessed by a program, `HDFPerl::h5fclose_p` must be called to release the resources used by the file. This call is mandatory.

```
status HDFPerl::h5fclose_p(file_id)
```

- The root group is automatically created when a file is created. Every file has a root group and the path name of the root group is always `/`.

- `HDFPerl::h5gcreate_p` creates a new empty group.

```
group_id HDFPerl::h5gcreate_p(loc_id, name, size_hint)
```

  - The *loc_id* parameter specifies the location at which to create the group.
  - The *name* parameter specifies the name of the group to be created.

- The *size_hint* parameter specifies how much file space to reserve to store the names that will appear in the group. Passing a value of zero is usually adequate since the library is able to dynamically resize the name heap.
- The function returns a valid group identifier if successful and a negative value otherwise.

- `HDFPerl::h5gclose_p` closes the group. This call is mandatory.

```
status HDFPerl::h5gclose_p(group_id)
```

- `HDFPerl::h5screate_p` creates a new simple dataspace and returns a dataspace identifier.

```
dspace_id HDFPerl::h5screate_p(rank, dims)
```

  - The *rank* parameter specifies the rank, i.e., the number of dimensions, of the dataset.
  - The *dims* parameter is a reference for an array that specifies the size of each dimension of the dataset.
  - The function returns the dataspace identifier if successful; otherwise it returns a negative value.

- `HDFPerl::h5sclose_p` closes the dataspace.

```
status HDFPerl::h5sclose_p(dspace_id)
```

- `HDFPerl::h5dcreate_p` creates a dataset at the specified location and returns a dataset identifier.

```
dset_id HDFPerl::h5dcreate_p(loc_id, name, type_id,
      dspace_id, creation_plist)
```

  - The *loc_id* parameter is the location identifier.
  - The *name* parameter is the name of the dataset to create.
  - The *type_id* parameter specifies the datatype identifier.
  - The d*space_id* parameter is the dataspace identifier.
  - The *creation_plist* parameter specifies the dataset creation property list. `$Init::H5P_DEFAULT` specify the default dataset creation property list.
  - The function returns the dataset identifier if successful and a negative value otherwise.

- `HDFPerl::h5dcreate_p` creates an empty array and initializes the data to 0.

- When a dataset is no longer accessed by a program, `HDFPerl::h5dclose_p` must be called to release the resource used by the dataset. This call is mandatory.

```
status HDFPerl::h5dclose_p(dset_id)
```

- `HDFPerl::h5dwrite_`*`datatype`*`_p` writes raw data from an application buffer to the specified dataset. *datatype* in the name of the function should be replaced with `double, float, int, int8, string,` or `vlstring` as appropriate.

```
status HDFPerl::h5dwrite_datatype_p(dset_id, mem_type_id,
      mem_dspace_id, file_dspace_id, transfer_plist, buf)
```

  - The *dset_id* is the dataset identifier.
  - The *mem_type_id* parameter is the identifier of the dataset's memory datatype. `$Init::H5T_NATIVE_INT` is an integer datatype for the machine on which the library was compiled.
  - The *mem_dspace_id* parameter is the identifier of the dataset's memory dataspace.
  - The *file_dspace_id* parameter is the identifier of the dataset's file dataspace.
  - The *transfer_plist* parameter is the data transfer property list identifier. `$Init::H5P_DEFAULT` is the default value and indicates that the default data transfer property list is used.
  - The *buf* parameter is a reference for the data buffer array to write.
  - This function returns a non-negative value if successful; otherwise it returns a negative value.

- `HDFPerl::h5dread_`*`datatype`*`_p` reads raw data from the specified dataset to an application buffer. *datatype* in the name of the function should be replaced with `double, float, int, int8, string,` or `vlstring` as appropriate.

```
buf_ref HDFPerl::h5dread_datatype_p(dset_id, mem_dspace_id,
      file_dspace_id, transfer_plist)
```

  - The *dset_id* is the dataset identifier.
  - The *mem_dspace_id* parameter is the identifier of the dataset's memory dataspace.
  - The *file_dspace_id* parameter is the identifier of the dataset's file dataspace.
  - The *transfer_plist* parameter is the data transfer property list identifier. `$Init::H5P_DEFAULT` is the default value and indicates that the default data transfer property list is used.
  - The function returns a reference for the reading buffer.

## *Using Compound Datatypes*

A compound datatype is similar to the definition of a database record. It is a collection of zero or more datatypes and can include nested compound datatypes. To create and use a compound datatype you need to be familiar with some properties of the compound datatype:

- It has a fixed total size, in bytes.
- Each member (or field) has its own datatype.
- Each member has a fixed byte offset, which locates the first byte (smallest byte address) of that member in the compound datatype.

Compound datatypes must be built out of other datatypes. First, one creates an empty compound datatype and specifies its total size. Then members are added to the compound datatype in any order.

A programming example is presented in List 2 showing how to use compound datatypes. It creates a one-dimensional array of 5 elements. Each element is a compound datatype formed by two members: *id* is a 7-char string, and *value* is an integer. The memory buffer containing data to be written has the following structure:

| id | value |
|---|---|
| “id1” | 10 |
| “id2” | 20 |
| “id3” | 30 |
| “id4” | 40 |
| “id5” | 50 |

Currently, this table in memory cannot be written to a dataset using a single I/O operation. Each field set (or column) must be written separately using a dummy compound datatype. Such datatypes must be created temporarily with only the member to be written.

```
#!/usr/bin/perl

# load appropriate modules
use lib "/install_path";
use HDFPerl;
use strict;
use Init;

# initialize HDF5 constants
Init::initialize();

# create a new HDF File.
my $fid = HDFPerl::h5fcreate_p("file1.h5", $Init::H5F_ACC_TRUNC,
         $Init::H5P_DEFAULT, $Init::H5P_DEFAULT);
```

```
# create a datatype formed by string of 7 chars. Note that size
# is 7 bytes.
my $stid = HDFPerl::h5tcreate_string_p(7);

# get the size in bytes of an integer
my $intsize = HDFPerl::h5tget_size_p($Init::H5T_NATIVE_INT);

# create a compound datatype that includes a 7-char string, and an
# integer
my $tid = HDFPerl::h5tcreate_compound_p(7+$intsize);
HDFPerl::h5tinsert_p($tid, "id", 0, $stid);
HDFPerl::h5tinsert_p($tid, "value", 7, $Init::H5T_NATIVE_INT);

# creates a dataspace: a one-dimensional array of length 5
my @dims=(5);
my $sid = HDFPerl::h5screate_p(1, \@dims);

# creates a dataset with the specified dataspace. Each element is
# an instance of the compound datatype
my $did = HDFPerl::h5dcreate_p($fid, "dataset", $tid, $sid,
      $Init::H5P_DEFAULT);

# create a compound structure in memory
my @id=("id1","id2","id3","id4","id5");
my @value=(10,20,30,40,50);
my @buffer=(\@id, \@value);

# write "id" column. Note the creation of a dummy compound datatype
# containing only the member "id" at offset 0
my $ctid = HDFPerl::h5tcreate_compound_p(7);
HDFPerl::h5tinsert_p($ctid, "id", 0, $stid);
HDFPerl::h5dwrite_string_p($did, $ctid, $sid, $sid, $Init::H5P_DEFAULT,
      $buffer[0]);
HDFPerl::h5tclose_p($ctid);

# write "value" column. Note the creation of a dummy compound
# datatype containing only the member "value" at offset 0
my $ctid = HDFPerl::h5tcreate_compound_p($intsize);
HDFPerl::h5tinsert_p($ctid, "value", 0, $Init::H5T_NATIVE_INT);
HDFPerl::h5dwrite_int_p($did, $ctid, $sid, $sid, $Init::H5P_DEFAULT,
      $buffer[1]);
HDFPerl::h5tclose_p($ctid);

# close resources
HDFPerl::h5tclose_p($stid);
HDFPerl::h5tclose_p($tid);
HDFPerl::h5dclose_p($did);
HDFPerl::h5sclose_p($sid);
HDFPerl::h5fclose_p($fid);
```

**List 2 Script showing use of compound datatypes**

As before, we can use *h5dump* to visualize the contents of the HDF5 file:

```
HDF5 "file1.h5" {
GROUP "/" {
   DATASET "dataset" {
      DATATYPE  H5T_COMPOUND {
         H5T_STRING {
            STRSIZE 7;
            STRPAD H5T_STR_NULLTERM;
            CSET H5T_CSET_ASCII;
            CTYPE H5T_C_S1;
         } "id";
         H5T_STD_I32LE "value";
      }
      DATASPACE  SIMPLE { ( 5 ) / ( 5 ) }
      DATA {
      (0): {
            "id1",
            10
         },
      (1): {
            "id2",
            20
         },
      (2): {
            "id3",
            30
         },
      (3): {
            "id4",
            40
         },
      (4): {
            "id5",
            50
         }
      }
   }
}
}
```

## Remarks

- `HDFPerl::h5tcreate_compound_p` creates a new compound datatype with the specified number of bytes.

  ```
  type_id HDFPerl::h5tcreate_compound_p(size)
  ```

  - The *size* parameter specifies the number of bytes in the datatype to create.

- `HDFPerl::h5tinsert_p` adds a member to the compound datatype specified by *type_id*.

```
status HDFPerl::h5tinsert_p(type_id, name, offset,
      member_id)
```

  - The *type_id* parameter is the identifier of the compound datatype to modify.
  - The *name* parameter is the name of the member to insert. The new member name must be unique within a compound datatype.
  - The *offset* parameter is the offset in the memory structure of the member to insert.
  - The *member_id* parameter is the datatype identifier of the member to insert.

- `HDFPerl::h5tclose_p` releases a datatype.

```
status HDFPerl::h5tclose_p(type_id)
```

- The *type_id* parameter is the identifier of the datatype to release.

## *Extendible Datasets and Compressed Storage*

When creating a dataset, HDF5 allows the user to specify how raw data is organized and/or compressed on disk. This information is stored in a dataset creation property list and passed to the dataset interface. The raw data on disk can be stored in several ways, including contiguously (in the same linear way that it is organized in memory), partitioned into chunks, or stored externally.

The application of chunked storage permits the creation of a dataset whose dimensions can grow, i.e. an extendible dataset. HDF5 allows an application to define the dimensions of such a dataset to have certain initial sizes, then later to increase the sizes of any of the dimensions.

An additional advantage of using a chunked layout on a dataset is that the dataset can then be compressed. A dataset can be compressed during creation by invoking the function `HDFPerl::h5pset_deflate_p`.

The programming example shown in List 3 creates a dataset as a one-dimensional array of 15 integers to be stored in chunks of 5 elements each[2]. This example specifies unlimited maximum dimensions and applies compression on the dataset. A memory

[2] Note: This chunk size is for illustrative purposes only. I/O performance can be affected greatly by the size and shape of a chunk. Chunk size and shape should be chosen to accommodate anticipated access patterns. Except in unusual circumstances, a chunk size should be at least a few thousand bytes.

buffer containing the data is defined as a one-dimensional array of 20 elements. The dataset size is extended to accommodate the buffer size, and the write operation is performed.

```
#!/usr/bin/perl

# load appropriate modules
use lib "/install_path";
use HDFPerl;
use strict;
use Init;

# initialize HDF5 constants
Init::initialize();

# create a new HDF File.
my $fid = HDFPerl::h5fcreate_p("file2.h5", $Init::H5F_ACC_TRUNC,
         $Init::H5P_DEFAULT, $Init::H5P_DEFAULT);

# create an extendible one-dimensional dataspace with a current size
# of 15, and unlimited maximum dimensions
my @cur_dims=(15);
my @max_dims=($Init::H5S_UNLIMITED);
my $sid = HDFPerl::h5screate_simple_p(1, \@cur_dims, \@max_dims);

# create property list for dataset creation. Chunking and compression
# settings will be applied to this property list.
my $pid = HDFPerl::h5pcreate_p($Init::H5P_DATASET_CREATE);

# set chunking with each chunk equal to a one-dimensional array of 5
# elements
my @chk_dims=(5);
HDFPerl::h5pset_chunk_p($pid, 1, \@chk_dims);

# set storage compression level to 7
HDFPerl::h5pset_deflate_p($pid, 7);

# create extendible dataset using the defined dataspace and property
# list
my $did = HDFPerl::h5dcreate_p($fid, "dataset", $Init::H5T_NATIVE_INT,
      $sid, $pid);

# close dataset creation property list
HDFPerl::h5pclose_p($pid);

# define writing buffer of 20 integers
my @buffer=();
my $i;
for ($i = 0; $i < 20; $i++){
      $buffer[$i]=$i+1;
}

# dataset must be extended so that the buffer can fit in
$cur_dims[0] = (20);
HDFPerl::h5dextend_p($did, \@cur_dims);
```

```
# close outdated dataspace and get an updated one from the dataset
HDFPerl::h5sclose_p($sid);
$sid = HDFPerl::h5dget_space_p($did);

# write buffer into the dataset
HDFPerl::h5dwrite_int_p($did, $Init::H5T_NATIVE_INT, $sid, $sid,
      $Init::H5P_DEFAULT, \@buffer);

# close resources
HDFPerl::h5sclose_p($sid);
HDFPerl::h5dclose_p($did);
HDFPerl::h5fclose_p($fid);
```

**List 3 Script showing extendible and compressed datasets**

## Remarks

- `HDFPerl::h5screate_simple_p` creates a new simple dataspace and returns a dataspace identifier.

  ```
  dspace_id HDFPerl::h5screate_simple_p(rank, dims, maxdims)
  ```

  - The *rank* parameter specifies the rank, i.e., the number of dimensions, of the dataset.
  - The *dims* parameter is a reference for an array that specifies the size of each dimension of the dataset.
  - The *maxdims* is a reference for an array that specifies the upper limit on the size of each dimension of the dataset. If an element of the array is `$Init::H5S_UNLIMITED`, then the corresponding dataset dimension can be extended arbitrarily.
  - The function returns the dataspace identifier if successful; otherwise it returns a negative value.

- The routine `HDFPerl::h5pcreate_p` creates a new property list as an instance of a property list class. The signature is as follows:

  ```
  plist_id HDFPerl::h5pcreate_p(classtype)
  ```

  - The parameter *classtype* is the type of property list to create. Since in this document we only use dataset creation property lists, the class type utilized is `$Init::H5P_DATASET_CREATE`.
  - The property list identifier is returned if successful; otherwise a negative value is returned.

- The routine `HDFPerl::h5pset_chunk_p` sets the size of the storage chunks used to store a chunked layout dataset. The signature of this routine is as follows:

```
status HDFPerl::h5pset_chunk_p(plist_id, ndims, dims)
```

  - The *plist_id* parameter is the identifier for the dataset creation property list.
  - The *ndims* parameter is the number of dimensions of each chunk. Currently, this parameter must be the same as the rank of the dataset.
  - The *dims* parameter is a reference for an array containing the chunk size on each dimension.
  - A non-negative value is returned if successful; otherwise a negative value is returned.

- The `HDFPerl::h5dextend_p` routine extends a dataset dimensions up to their maximum sizes. The signature is as follows:

```
status HDFPerl::h5dextend_p(dset_id, size)
```

  - The *dset_id* parameter is the dataset identifier.
  - The *size* parameter is a reference for an array containing the new magnitude of each dimension.
  - This function returns a non-negative value if successful and a negative value otherwise.

- The routine `HDFPerl::h5pset_deflate_p` sets compression method and level for a new dataset. The signature is as follows:

```
status HDFPerl::h5pset_deflate_p(plist_id, level)
```

  - The *plist_id* parameter is the dataset creation property list identifier.
  - The *level* parameter, is the compression level, which should be a value from zero to nine, inclusive.
  - Lower compression levels are faster but result in less compression.
  - This function returns a non-negative value if successful and a negative value otherwise.

- The `HDFPerl::h5pclose_p` routine terminates access to a property list. The signature is as follows:

```
status HDFPerl::h5pclose_p(plist_id)
```

## *Partial Writing to a Dataset*

We have presented programming examples that write a dataset in its entirety. However, writing into a portion of a dataset is also possible by selecting the region of interest (or hyperslab). A hyperslab selection can be a logically contiguous collection of array elements in a dataspace, or it can be a regular pattern of elements or blocks in a dataspace. One can select a hyperslab to write to or read from with the function `HDFPerl::h5sselect_hyperslab_p`.

The programming example shown in List 4 creates a dataset formed by a 10-element integer array, and a memory buffer consisting of a 7-element integer array (to be written partially in the dataset). It selects a 3-element hyperslab from the memory dataspace starting at offset 2, and performs a selection of the same size in the file dataspace starting at offset 5 (recall that Perl arrays start with index zero). Since the selections are contiguous, the specified selection stride is 1. Finally, the memory selection is written into the dataset selection.

**7-element memory array:**


**10-element file dataset:**


```
#!/usr/bin/perl

# load appropriate modules
use lib "/install_path";
use HDFPerl;
use strict;
use Init;

# initialize HDF5 constants
Init::initialize();
# create a new HDF File.
my $fid = HDFPerl::h5fcreate_p("file3.h5", $Init::H5F_ACC_TRUNC,
         $Init::H5P_DEFAULT, $Init::H5P_DEFAULT);

# creates the file dataspace: an array of length 10
my @dims=(10);
my $sid = HDFPerl::h5screate_p(1, \@dims);

# creates a dataset in the file with the specified dataspace. Each
# element is an integer
my $did=HDFPerl::h5dcreate_p($fid, "dataset", $Init::H5T_NATIVE_INT,
      $sid, $Init::H5P_DEFAULT);
```

```
# creates the memory dataspace: an array of length 7
my @mdims=(7);
my $mid = HDFPerl::h5screate_p(1, \@mdims);

# create a memory array of length 7
my @buffer=(1, 2, 3, 4, 5, 6, 7);
# defines parameters for selection on the memory dataspace
my @unitarray=(1);
my @memoffset=(2);
my @memlength=(3);

# performs the selection in memory dataspace
HDFPerl::h5sselect_hyperslab_p($mid, $Init::H5S_SELECT_SET,
      \@memoffset, \@unitarray, \@memlength, \@unitarray);

# defines parameters for selection on the file dataspace
my @dsetoffset=(5);
my @dsetlength=(3);

# performs the selection in file dataspace
HDFPerl::h5sselect_hyperslab_p($sid, $Init::H5S_SELECT_SET,
      \@dsetoffset, \@unitarray, \@dsetlength, \@unitarray);

# writes memory selection into dataset selection. Note the use of
# memory and file dataspaces
HDFPerl::h5dwrite_int_p($did, $Init::H5T_NATIVE_INT, $mid, $sid,
      $Init::H5P_DEFAULT, \@buffer);

# close resources
HDFPerl::h5sclose_p($sid);
HDFPerl::h5sclose_p($mid);
HDFPerl::h5dclose_p($did);
HDFPerl::h5fclose_p($fid);
```

**List 4 Sample script showing use of hyperslab selection**

Please refer to the HDF5 Users' Guide and Reference Manual for detailed information regarding hyperslab selections.

## Remarks

- `HDFPerl::h5sselect_hyperslab_p` selects a hyperslab region to add to the current selected region for a specified dataspace.

```
status HDFPerl::h5sselect_hyperslab_p(dspace_id, operator,
      start, stride, count, block)
```

  - The parameter *dspace_id* is the dataspace identifier for the specified dataspace.
  - The parameter *operator* can be set to one of the following:
    - `$Init::H5S_SELECT_SET`

        - Replace the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
    - `$Init::H5S_SELECT_OR`
        - Add the new selection to the existing selection.
- The *start* parameter is a reference for an array that determines the starting coordinates of the hyperslab to select.
- The *stride* parameter is a reference for an array that indicates which elements along a dimension are to be selected by skipping the specified stride. A contiguous selection has a stride equal to 1.
- The *count* parameter is a reference for an array that determines how many blocks to select from the dataspace in each dimension.
- The *block* parameter is a reference for an array that determines the size of the element block used in the selection. This parameter is used in conjunction with the *count* parameter.
- The *start*, *stride*, *count*, and *block* arrays must be the same size as the rank of the dataspace.
- A non-negative value is returned if successful, and a negative value otherwise.

## *Appendix: Migrating data in FASTA format to HDF5 files*

FASTA is a text-based file format used in bioinformatics to describe nucleic acid or peptide sequences. A FASTA sequence record from the National Center for Biotechnology Information (NCBI) has the following form:

```
>gi|id1|gb|id2|id3 Comment about sequence
GATAATGGTA
```

A sequence record starts with a ">" followed by one or more identifiers for the particular sequence (keywords like "gi", "gb", and separators like "|" are not considered). After the first blank space, the rest of the line is considered to be a comment or description about the sequence. Actual sequence data starts in the next line and usually takes in several lines. In general, a FASTA file contains numerous sequence records.

The sequence data in FASTA format can be migrated to HDF5 files using the Perl script *fasta2hdf5.pl* as follows:

```
fasta2hdf5.pl  fasta_file  hdf5_file
```

The script organizes the sequence record data into three HDF5 datasets: *sequences*, *ids*, and *comments*. The *sequences* dataset is an array of characters that appends the base sequences read from the FASTA file one after another.

Given that the *ids* and *comments* datasets are one-dimensional arrays of compound datatypes, they resemble tables of a database with the following description:

***ids*:**

*id* : identifier for a particular sequence.
*index* : location of the sequence information in the *comments* dataset.

***comments*:**

*comment* : generic description about the sequence
*offset* : location of the sequence data in the *sequences* dataset
*length* : length of sequence data

The next sample sequence record would be stored in an HDF5 file as shown in Figure A.1.

```
>gi|id1|gb|id2|id3 Comment about sequence
GATAATGGTA
```

***IDS* DATASET**

| array index | id | index |
|---|---|---|
| … | … | … |
| … | id1 | i |
| … | id2 | i |
| … | id3 | i |
| … | … | … |

***COMMENTS* DATASET**

| array index | comment | offset | length |
|---|---|---|---|
| … | … | … | … |
| i | Comment about sequence | j | 10 |
| … | … | … | … |

***SEQUENCES* DATASET**

| array index | data element |
|---|---|
| … | … |
| j | G |
| j+1 | A |
| j+2 | T |
| j+3 | A |
| j+4 | A |
| j+5 | T |
| j+6 | G |
| j+7 | G |
| j+8 | T |
| j+9 | A |
| … | … |

**Figure A.1 Sequence record data in HDF5 file**

The following is a high level description of the *fasta2hdf5.pl* script

```
create HDF5 file.
create sequences dataset in the HDF5 file, and set writing offset
to zero.
set record_counter to zero.
for each sequence record in FASTA file
      read the ids and store them along with the record_counter
      in the ids_hash buffer.
      read the comment and base sequence.
      compute length of base sequence.
      store comment, offset, and length in the comments_buf
      buffer.
      extend sequences dataset to accommodate base sequence.
      write base sequence into the sequences dataset at location
      referenced by offset.
      update offset for next base sequence.
      increase record_counter.
end

sort ids_hash with respect to ids.
copy sorted ids_hash into ids_buf buffer.
create ids dataset in the HDF5 file.
write ids_buf buffer into ids dataset.

create comments dataset in the HDF5 file.
write  comments_buf buffer into comments dataset.
```

The code listing for the *fasta2hdf5.pl* script is the following:

```
#!/usr/bin/perl

# This script reads a FASTA file and writes an hdf5 file with 3
# datasets:
# ids is an ordered dataset with sequence id as key and a pointer to
# comments dataset.
# comments dataset contains the comment, offset and length information
# about sequences stored in the sequences dataset
# sequences dataset stores all the sequences in a single array

use lib "/home1/HDFPerl/lib";
use HDFPerl;
use strict;
use Init;

Init::initialize();

if ($#ARGV+1 < 2){
    print "\nUsage:\n";
    print "\t./fasta2hdf5.pl fasta_file hdf5_file\n\n";
    exit(1);
}
```

```
my $input_filename = $ARGV[0];
my $hdf_filename=$ARGV[1];
my %ids_hash = ();
my @comments_buf = ();
my @unitarray=(1);
my @h5offset=(0);
my @h5length=();
my $compression_level=7; # from 0 to 9
my $i=0;
my @dims=();

# create a new HDF File.
my $fid = HDFPerl::h5fcreate_p($hdf_filename, $Init::H5F_ACC_TRUNC,
         $Init::H5P_DEFAULT, $Init::H5P_DEFAULT);

# create chunked and compressed "sequences" dataset
my @cur_dims=(1);
my @chk_dims=(5000);
my @maxdims=($Init::H5S_UNLIMITED);
my $sid = HDFPerl::h5screate_simple_p(1, \@cur_dims, \@maxdims);
my $pid = HDFPerl::h5pcreate_p($Init::H5P_DATASET_CREATE);
HDFPerl::h5pset_chunk_p($pid, 1, \@chk_dims);
HDFPerl::h5pset_deflate_p($pid, $compression_level);
my $buf_name="sequences";
my $did = HDFPerl::h5dcreate_p($fid, $buf_name, $Init::H5T_NATIVE_CHAR,
                $sid, $pid);
HDFPerl::h5pclose_p($pid);

open(FILE, "< $input_filename");
my $line = <FILE>;

# iterate over every record
while ($line){
    chomp $line;
    my @header=();
    my @ids=();
    @header=split(/ /, $line);

    # string starting with ">" and ending with first " " is splitted at
    # "|" to form the array @ids. IDs are extracted at
    # locations 1, 3, and 4 and stored in the %ids_hash data structure
    @ids=split(/\|/, $header[0]);
    $ids_hash{$ids[1]}=$i;
    $ids_hash{$ids[3]}=$i;
    $ids_hash{$ids[4]}=$i;

    # rest of string is stored in $comments
    my $comment=join(' ',@header[1 .. $#header]);
    my $j=0;
    my @seq=();
    $seq[$j] = <FILE>;

    # iterate over every line of a sequence
    while ( ($seq[$j] !~ /^>/) && ($seq[$j]) ){
        chomp $seq[$j];
        $j++;
        $seq[$j] = <FILE>;
```

```
    }

    # store record information in @comments_buf data structure
    my $sequence = join('',@seq[0 .. $j-1]);
    @h5length = (length($sequence));
    $comments_buf[0][$i] = $comment;
    $comments_buf[1][$i] = $h5offset[0];
    $comments_buf[2][$i] = $h5length[0];

    $cur_dims[0] = $h5offset[0]+$h5length[0];
    HDFPerl::h5dextend_p($did, \@cur_dims);
    HDFPerl::h5sclose_p($sid);
    $sid = HDFPerl::h5dget_space_p($did);

    # selects location and length on the dataset space for writing
    HDFPerl::h5sselect_hyperslab_p($sid, $Init::H5S_SELECT_SET,
      \@h5offset, \@unitarray, \@h5length, \@unitarray);
    my $mid = HDFPerl::h5screate_simple_p(1, \@h5length, \@h5length);
    my @post_sequence = split(//, $sequence);

    # writes into the dataset
    HDFPerl::h5dwrite_string_p($did, $Init::H5T_NATIVE_CHAR, $mid,
      $sid,$Init::H5P_DEFAULT, \@post_sequence);
    HDFPerl::h5sclose_p($mid);

    # settings for next iteration
    $h5offset[0]=$h5offset[0]+$h5length[0];
    $i++;
    $line=$seq[$j];
}
my $num_recs = $i;
HDFPerl::h5dclose_p($did);
HDFPerl::h5sclose_p($sid);

my $id;
my $j = 0;
my @ids_buf;

# sorts %ids_hash and prepares @ids_buf data structure for writing
foreach $id (sort keys %ids_hash){
    $ids_buf[0][$j]=$id;
    $ids_buf[1][$j]=$ids_hash{$id};
    $j++;
}
my $num_ids = $j;

# create compound datatype for "ids" dataset
my $stid = HDFPerl::h5tcreate_string_p(10);
my $intsize = HDFPerl::h5tget_size_p($Init::H5T_NATIVE_INT);
my $tid = HDFPerl::h5tcreate_compound_p(10+$intsize);
HDFPerl::h5tinsert_p($tid, "id", 0, $stid);
HDFPerl::h5tinsert_p($tid, "index", 10, $Init::H5T_NATIVE_INT);

# create chunked and compressed "ids" dataset
@dims=($num_ids);
@chk_dims=(int($num_ids/10)+1000);
$sid = HDFPerl::h5screate_simple_p(1, \@dims, \@maxdims);
```

```
$pid = HDFPerl::h5pcreate_p($Init::H5P_DATASET_CREATE);
HDFPerl::h5pset_chunk_p($pid, 1, \@chk_dims);
HDFPerl::h5pset_deflate_p($pid, $compression_level);
my $did = HDFPerl::h5dcreate_p($fid, "ids", $tid,
                $sid, $pid);

# write field "id" data into "ids" dataset
my $mtid = HDFPerl::h5tcreate_compound_p(10);
HDFPerl::h5tinsert_p($mtid, "id", 0, $stid);
HDFPerl::h5dwrite_string_p($did, $mtid, $sid, $sid, $Init::H5P_DEFAULT,
    $ids_buf[0]);
HDFPerl::h5tclose_p($mtid);

# write field "index" data into "ids" dataset
$mtid = HDFPerl::h5tcreate_compound_p($intsize);
HDFPerl::h5tinsert_p($mtid, "index", 0, $Init::H5T_NATIVE_INT);
HDFPerl::h5dwrite_int_p($did, $mtid, $sid, $sid, $Init::H5P_DEFAULT,
    $ids_buf[1]);
HDFPerl::h5tclose_p($mtid);

# close resources
HDFPerl::h5dclose_p($did);
HDFPerl::h5sclose_p($sid);
HDFPerl::h5pclose_p($pid);

# create compound datatype for "comments" dataset
my $vlstid = HDFPerl::h5tcreate_string_p($Init::H5T_VARIABLE);
my $vlsize = HDFPerl::h5tget_size_p($vlstid);
my $tid = HDFPerl::h5tcreate_compound_p($vlsize+2*$intsize);
HDFPerl::h5tinsert_p($tid, "comment", 0, $vlstid);
HDFPerl::h5tinsert_p($tid, "offset", $vlsize, $Init::H5T_NATIVE_INT);
HDFPerl::h5tinsert_p($tid, "length", $vlsize+$intsize,
$Init::H5T_NATIVE_INT);

# create chunked and compressed "comments" dataset
@dims=($num_recs);
@chk_dims=(int($num_recs/10)+1000);
$sid = HDFPerl::h5screate_simple_p(1, \@dims, \@maxdims);
$pid = HDFPerl::h5pcreate_p($Init::H5P_DATASET_CREATE);
HDFPerl::h5pset_chunk_p($pid, 1, \@chk_dims);
HDFPerl::h5pset_deflate_p($pid, $compression_level);
$did = HDFPerl::h5dcreate_p($fid, "comments", $tid, $sid,
       $pid);

# write field "comment" data into "comments" dataset
my $ctid = HDFPerl::h5tcreate_compound_p($vlsize);
HDFPerl::h5tinsert_p($ctid, "comment", 0, $vlstid);
HDFPerl::h5dwrite_vlstring_p($did, $ctid, $sid, $sid,
$Init::H5P_DEFAULT,
        $comments_buf[0]);
HDFPerl::h5tclose_p($ctid);

# write field "offset" data into "comments" dataset
my $ctid = HDFPerl::h5tcreate_compound_p($intsize);
HDFPerl::h5tinsert_p($ctid, "offset", 0, $Init::H5T_NATIVE_INT);
HDFPerl::h5dwrite_int_p($did, $ctid, $sid, $sid, $Init::H5P_DEFAULT,
       $comments_buf[1]);
```

```
HDFPerl::h5tclose_p($ctid);

# write field "length" data into "comments" dataset
my $ctid = HDFPerl::h5tcreate_compound_p($intsize);
HDFPerl::h5tinsert_p($ctid, "length", 0, $Init::H5T_NATIVE_INT);
HDFPerl::h5dwrite_int_p($did, $ctid, $sid, $sid, $Init::H5P_DEFAULT,
       $comments_buf[2]);
HDFPerl::h5tclose_p($ctid);

# close resources
HDFPerl::h5dclose_p($did);
HDFPerl::h5sclose_p($sid);
HDFPerl::h5pclose_p($pid);
HDFPerl::h5fclose_p($fid);
```